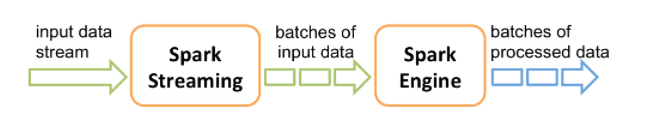
## 简介

Sparkstreaming是spark的一个分支，是主要用来处理批数据的。是一个高可用，容错的流处理框架之一。数据来源多种多样，像 Kafka, Flume, Kinesis, or TCP sockets等等，通过复杂的高阶函数将数据进行处理，可以流向到HDFS、Database、Dashboards(仪表盘)等等



sparkStreaming就收流数据或者批处理数据，然后将这些数据按照时间片进行分隔，发送给spark引擎进行处理，最终返回批处理数据



## sparkStreaming HelloWorld

1. 创建maven scala项目
2. 添加依赖

|  |
| --- |
| *<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-core -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-core\_2.11</**artifactId**>  <**version**>2.2.0</**version**> </**dependency**> *<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-streaming -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-streaming\_2.11</**artifactId**>  <**version**>2.2.0</**version**> </**dependency**> |

1. 编写代码，从网络端口获取数据，并统计单词出现的次数

|  |
| --- |
| **package** top.xiesen.bd14  **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.{Seconds, StreamingContext}  */\*\*  \* 从网络端口获取数据，并且统计每个单词出现的次数  \*/* **object** StreamingNetCatWordCount {  *// 在构建sparkStreaming程序的时候master，如果是local的话，那么local上必须保证有两个线程  // 如果使用local的话至少local[2],如果一个线程，那么只接受数据，不进行数据处理* **val** *conf* = **new** SparkConf().setMaster(**"local[\*]"**).setAppName(**"StreamingNetCatWordCount"**)  **val** *ssc* = **new** StreamingContext(*conf*, *Seconds*(3))   **def** main(args: Array[String]): Unit = {  *// 获取数据流* **val** dstream = *ssc*.socketTextStream(**"master"**, 9999)  **val** result = dstream.flatMap(x => x.split(**" "**))  .map(x => (x, 1))  .reduceByKey(\_ + \_)  result.print(20)  *ssc*.start()  *ssc*.awaitTermination()  } } |

1. 在linux上开启端口

|  |
| --- |
| nc -lk 9999 |

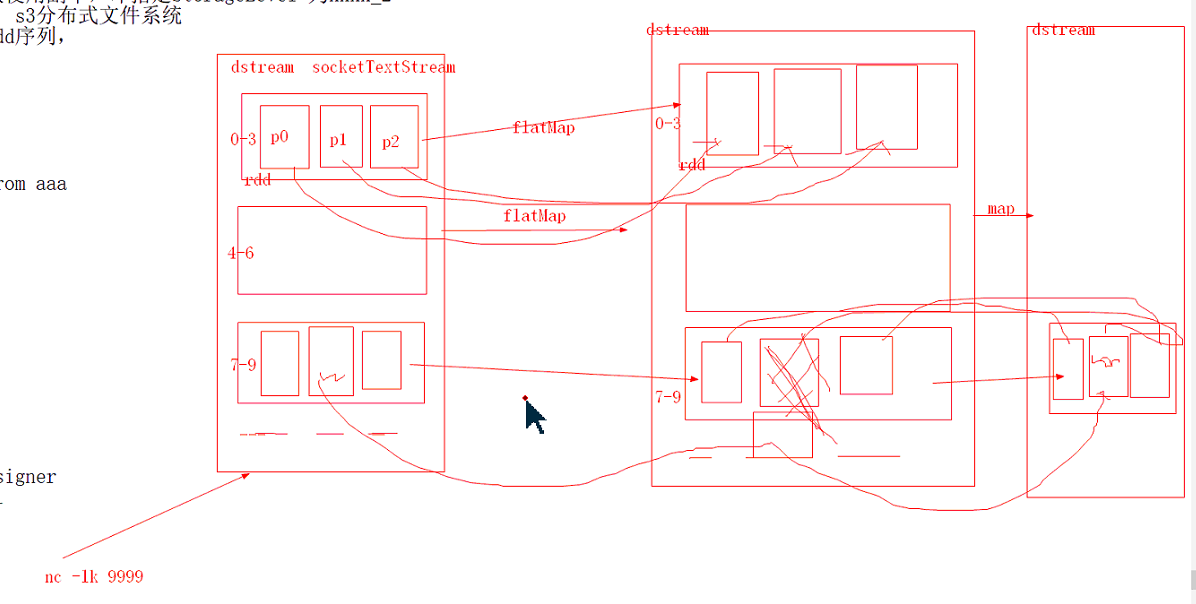
1. 运行程序
2. 在netcat上输入内容，以空格分开
3. 查看控制台输出的信息

## sparkStreaming代码书写流程

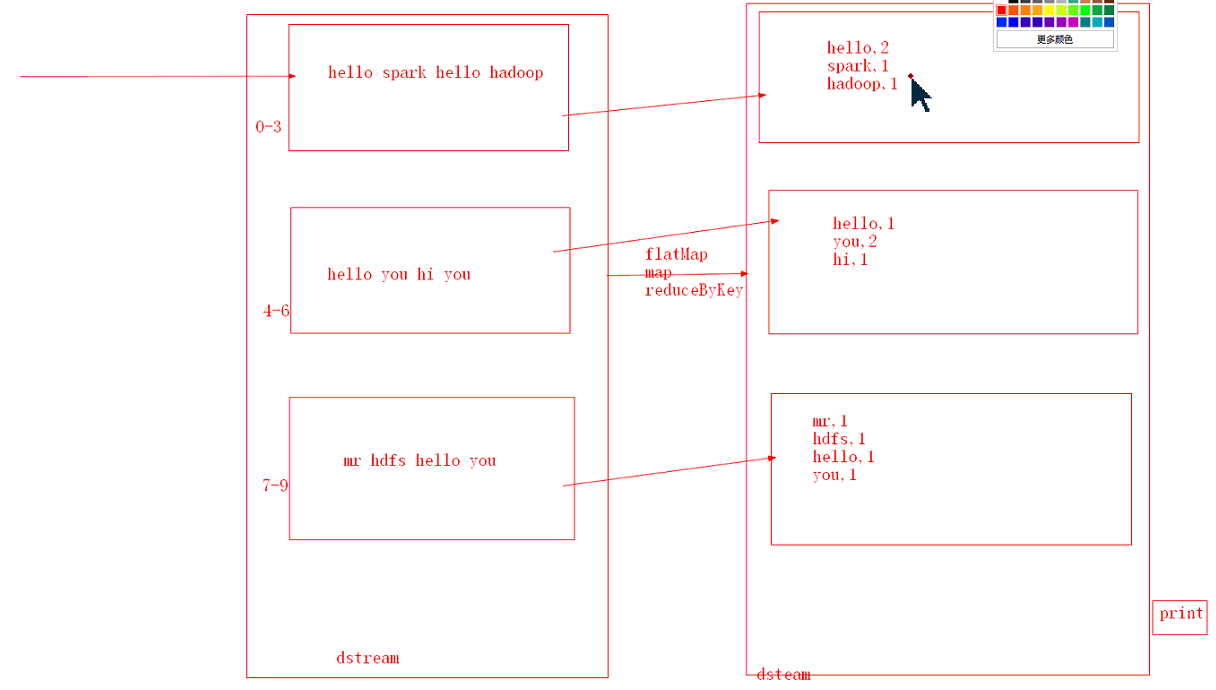
1. Sparkconf
2. 实例化StreamingContext实例
3. 使用StreamingContext获取DStream
4. DStreaming调用transformation和active进行操作
5. 对dstream处理的结果进行保存或输出
6. 调用StreamContext的start 启动和awaitTermination等待终结信号

## 容错机制

1. 构建dstream的时候使用副本，即指定StorageLeave为xxxxx\_2的形式时，可以进行容错，例如(MEMORY\_AND\_DISK\_SER\_2)
2. 对StreamingContext调用Checkpoint hdfs s3分布式系统
3. Dstream是离散的rdd序列，rdd的依赖关系构成dstream的依赖关系，当某个dstream挂了时，计算从该dstream的父dstream在新的节点上恢复后再继续计算



批次计算



## sparkStreaming累计计算方式

1. window窗口计算
2. State 状态更新
3. 其他自实现方式来累计计算

## sparkStreaming数据源

socketTextStream:从网络端口获取流数据转换成文本之后交由stream的dstream的transformation等方法处理

socketTextStream从网络端口获取数据流，不进行格式转换，形成字节数据的dStream

rawSocketStream直接从网络端口拿到数据，不经过转换形成dstream

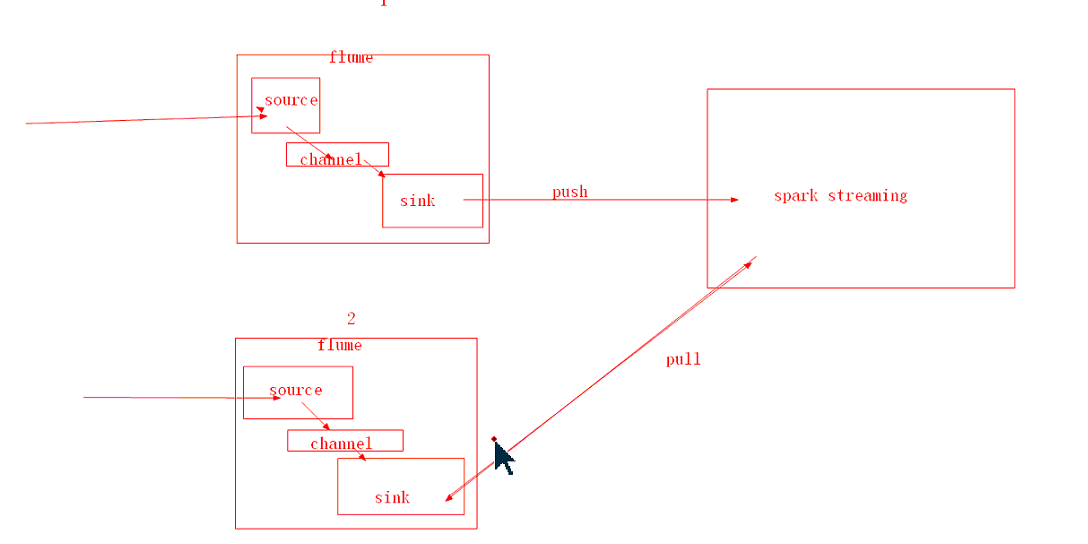
textFileStream监听hdfs的文件夹，当文件夹下有新文件的时候以文本的形式加载进来形成dstream

fileStream监听hdfs的文件夹，当文件夹下有新文件的时候以特定的格式(InputStream) 来加载数据形成dstream

queueStream把一个rdd队列对象，转换成dstream

## sparkStreaming与flume集成

sparkStreaming与flume集成的两种方式



### flume push数据到sparkStreaming形成dStream

* 1. 添加依赖

|  |
| --- |
| *<!--添加spark-flume-->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-streaming-flume\_2.11</**artifactId**>  <**version**>2.2.0</**version**> </**dependency**> |

* 1. 使用 val flumeStream = FlumeUtils.createStream(streamingContext, [chosen machine's hostname], [chosen port])

### flume pull数据到sparkStreaming形成dStream

* 1. 添加依赖
  2. 三个jar (spark-streaming-flume-sink\_2.11、scala-library、commons-lang3)，下载好放到flume的lib目录下
  3. Flume程序中的sink必须是org.apache.spark.streaming.flume.sink.SparkSink类型

|  |
| --- |
| agent.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink  agent.sinks.spark.hostname = <hostname of the local machine>  agent.sinks.spark.port = <port to listen on for connection from Spark>  agent.sinks.spark.channel = memoryChannel |

* 1. Streaming程序中获取flume数据流

|  |
| --- |
| **val** flumeDstream = FlumeUtils.createPollingStream(ssc,**"master"**,8888) |

方式二更加可靠，不过操作过程比较复杂，需要在每个节点上将2.b)中的jarcopy到相对应的位置

## SparkStreaming连接kafka

获取kafka的dstram

KafkaUtils.createDirectStream(ssc,LocationStrategies.PreferConsistent,ConsumerStrategy)

LocationStrategies：本地策略  
当exector集群节点可kafka的broker集群在同一个节点上，选择org.apache.spark.streaming.kafka010.PreferBrokers

当executor集群节点和kafka的集群节点相互独立的时，选择org.apache.spark.streaming.kafka010.PreferConsistent

ConsumerStrategy：消费者策略

消费者策略参数使用：ConsumerStrategy.Subscribe[String,String](topics,kafkaParams)

Topics:是消费者topic的名称的集合对象

kafkaParams：是一个map，填写kafka、consumer的配置参数

要保证Subscribe的两个泛型，和map的kv泛型保持一致

|  |
| --- |
| **package** top.xiesen.bd14.kafka  **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils, LocationStrategies} **import** org.apache.spark.streaming.{Duration, StreamingContext}  **object** KafkaDriverWordCount {   **val** *conf* = **new** SparkConf().setAppName(**"KafkaDriverWordCount"**).setMaster(**"local[\*]"**)  **val** *ssc* = **new** StreamingContext(*conf*, *Duration*(3000))   **def** main(args: Array[String]): Unit = {   **val** kafkaParams = *Map*[String, String](**"bootstrap.servers"** -> **"master:9092,slaver1:9092"** , **"key.deserializer"** -> **"org.apache.kafka.common.serialization.StringDeserializer"** , **"value.deserializer"** -> **"org.apache.kafka.common.serialization.StringDeserializer"** , **"group.id"** -> **"kafkatest"** , **"enable.auto.commit"** -> **"false"** )   **val** topics = *Array*(**"scalakafka"**)  **val** consumerStrategies = ConsumerStrategies.*Subscribe*[String, String](topics, kafkaParams)  **val** kafkaStream =  KafkaUtils.*createDirectStream*(*ssc*, LocationStrategies.*PreferConsistent*, consumerStrategies)   kafkaStream.map(x => x.value())  .flatMap(\_.split(**"\\s"**))  .map((\_, 1))  .reduceByKey(\_ + \_)  .print()  *ssc*.start()  *ssc*.awaitTermination()  } } |

## sparkStreaming的数据存储

### sparkStreaming将计算结果保存到mysql

saveAsobjectFiles

saveAsTextFiles

saveAsNewAPIHadoopFiles

saveAsHadoopFiles

这些文件保存成数据文件

更多生产环境下是要将数据存储在database中例如mysqll和hbase

Mysql和hbase都是使用foreachRDD

Dstream的foreachRDD方法，对rdd进行操作各自存储到mysql或hbase中

Rdd的数据保存到hbase

|  |
| --- |
| **import** java.sql.DriverManager  **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.{Seconds, StreamingContext} */\*\*  \* netcat word count 的结果保存在mysql中  \*/* **object** StreamingSave2Mysql {   **val** *conf* = **new** SparkConf().setMaster(**"local[\*]"**).setAppName(**"StreamingSave2Mysql"**)  **val** *ssc* = **new** StreamingContext(*conf*, *Seconds*(3))   **def** main(args: Array[String]): Unit = {  *// 获取流数据* **val** dstream = *ssc*.socketTextStream(**"master"**, 54444)  **val** result = dstream.flatMap(\_.split(**"\\s"**))  .map((\_, 1))  .reduceByKey(\_ + \_)   result.print()   *//把结果保存到mysql* result.foreachRDD((rdd, time) => {  **val** timestamp = time.milliseconds  *// rdd把数据保存到mysql中的过程* rdd.foreachPartition(wcs => {  Class.*forName*(**"com.mysql.jdbc.Driver"**)  **val** counection = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/xs"**, **"root"**, **"root"**)  **val** sql = **"insert into streaming\_wc (ts,word,wccount) values(?,?,?)"  val** prepareStatement = counection.prepareStatement(sql)  **for** (record <- wcs) {  prepareStatement.setLong(1, timestamp)  prepareStatement.setString(2, record.\_1)  prepareStatement.setInt(3, record.\_2)  prepareStatement.addBatch()  }  prepareStatement.executeBatch()  counection.close()  })  })   *ssc*.start()  *ssc*.awaitTermination()  } } |

### sparkStreaming将计算结果保存到hbase

|  |
| --- |
| **import** java.util **import** org.apache.hadoop.hbase.client.Put **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.{Seconds, StreamingContext} */\*\*  \* netcat word count 的结果保存在hbase中  \*/* **object** StreamingSave2Hbase {   **val** *conf* = **new** SparkConf().setAppName(**"StreamingSave2Hbase"**).setMaster(**"local[\*]"**)  **val** *ssc* = **new** StreamingContext(*conf*,*Seconds*(3))   **def** main(args: Array[String]): Unit = {  **val** dstream = *ssc*.socketTextStream(**"master"**,9999)  **val** result = dstream.flatMap(\_.split(**"\\s"**))  .map((\_,1))  .reduceByKey(\_ + \_)   *// 使用foreachRDD通过rdd的操作将数据保存在hbase中  // create 'streaming\_wc','i'  // 时间戳为rowkey，表示批次，然后word作为columequality count 作为value* result.foreachRDD((rdd,time) => {  **val** timestrap = time.milliseconds.toString  rdd.foreachPartition(wcs => {  *// 构建put集合* **val** table = HbaseUtils.*getTable*(**"streaming\_wc"**)  **val** putList = **new** util.ArrayList[Put]()  **for** ( wc <- wcs){  **val** put = **new** Put(timestrap.getBytes)  put.addColumn(**"i"**.getBytes(),wc.\_1.getBytes(),wc.\_2.toString.getBytes())  putList.add(put)  }  table.put(putList)  })  })   *ssc*.start()  *ssc*.awaitTermination()  } } |

## 窗口计算

countByValueAndWindow

countByWindow

reduceByWindow

window

groupByKeyAndWindow

reduceByKeyAndWindow

统计近一个小时的用户点击量，每5秒输出一次结果

窗口是一个时间段，而且是一个比微批量更大的一个时间段。因此窗口计算数据时多个批次累加数据量再进行计算。窗口函数的宽度必须是微批次宽度的整数倍

滑动宽度：两次计算之间的时间间隔，滑动的宽度必须是微批次的整数倍

计算的数据量由窗口的宽度来决定

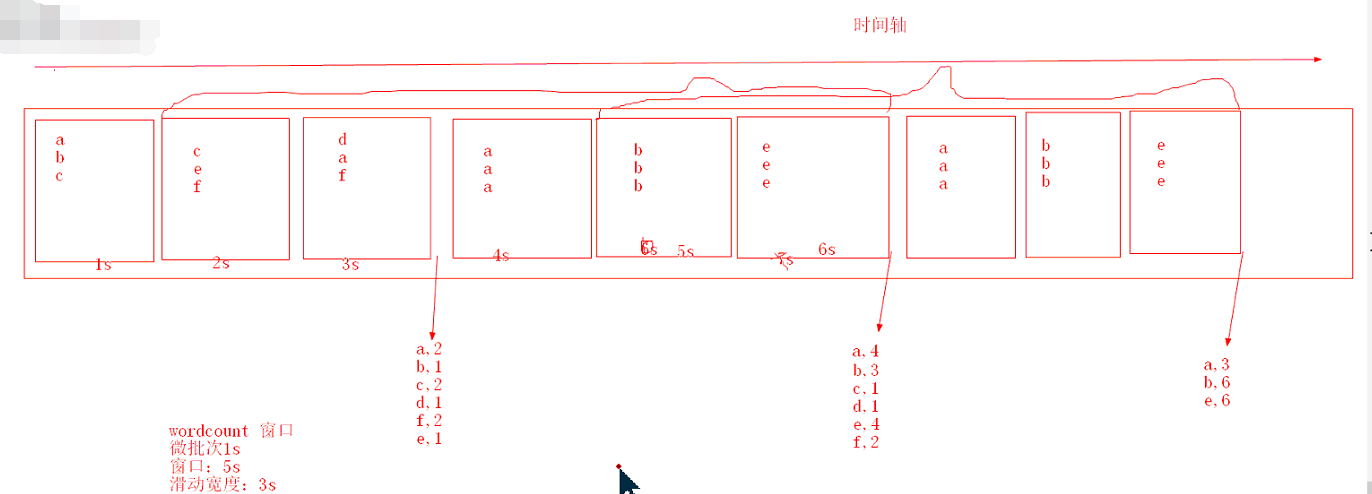
计算的频率(时间间隔)由滑动宽度来决定

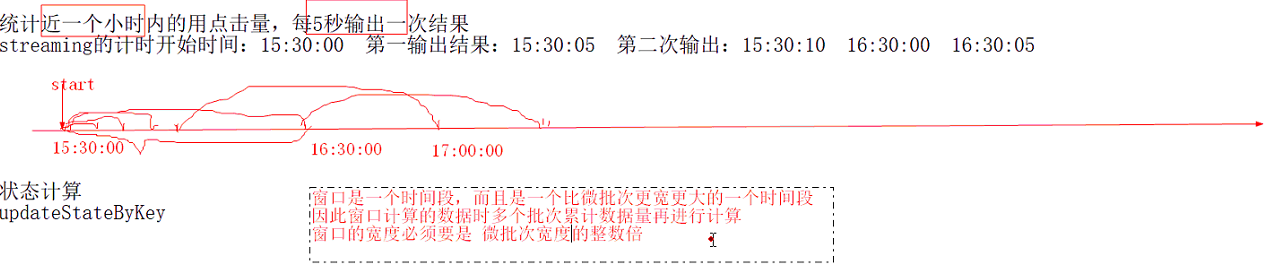
窗口是指时间段上的一个宽度，它用来划分 计算时参与计算的dStream中的多个微批次

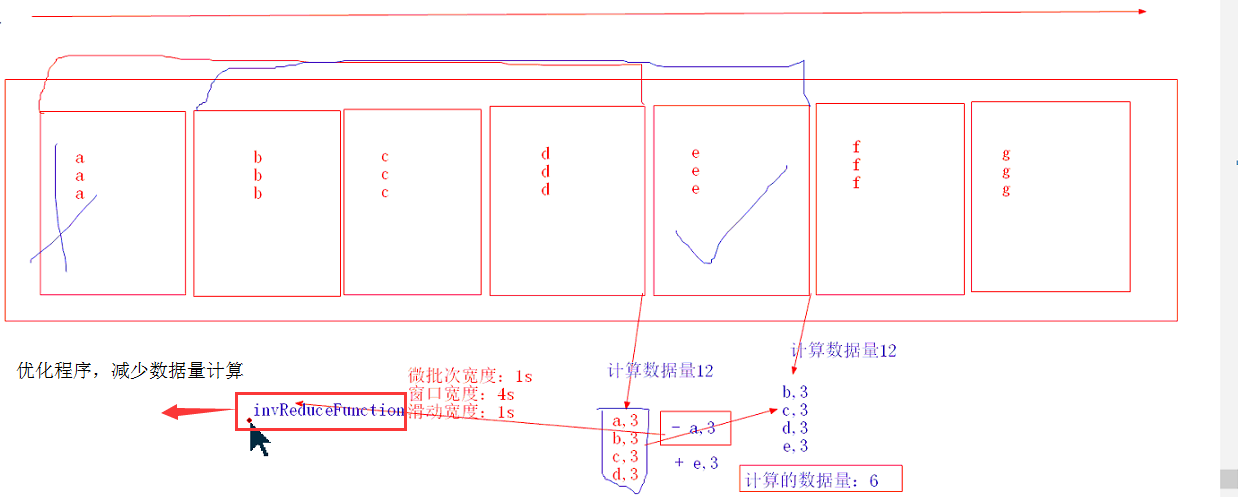
定义好窗口宽度后的dStream在进行计算时，所有窗口内的微批次都会参与计算，窗口宽度是一个笃定的时间长度，它必须是微批次的整数倍，每一个窗口计算的微批次的数量是一个定值

滑动宽度也是时间上的一个宽度，它用来指定sparkStreaming计算引擎两次执行计算之间的时间间隔，它必须是微批次的整数倍，微批次对于dStream来说是不可以再进行细分的

窗口计算中，窗口宽度决定着计算的数量，滑动宽度决定着计算的频率(两次计算的时间间隔)，而微批次的宽度决定着计算的数据量，滑动宽度决定着计算的频率(两次计算的时间间隔)，而微批次的宽度只代表接收器把多长时间段获取的数据生成一个rdd



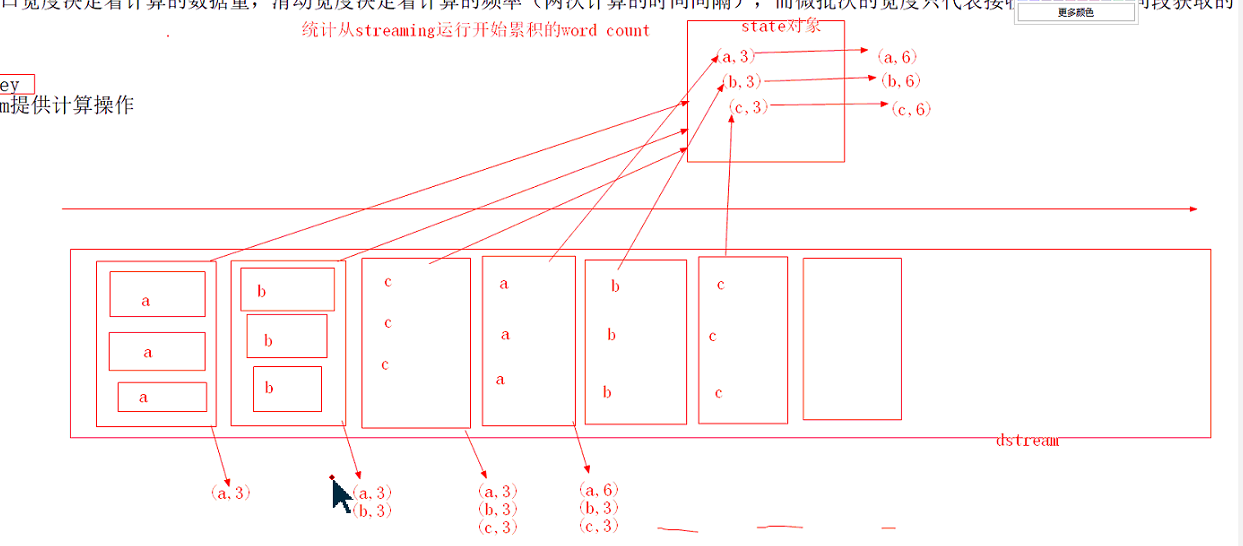


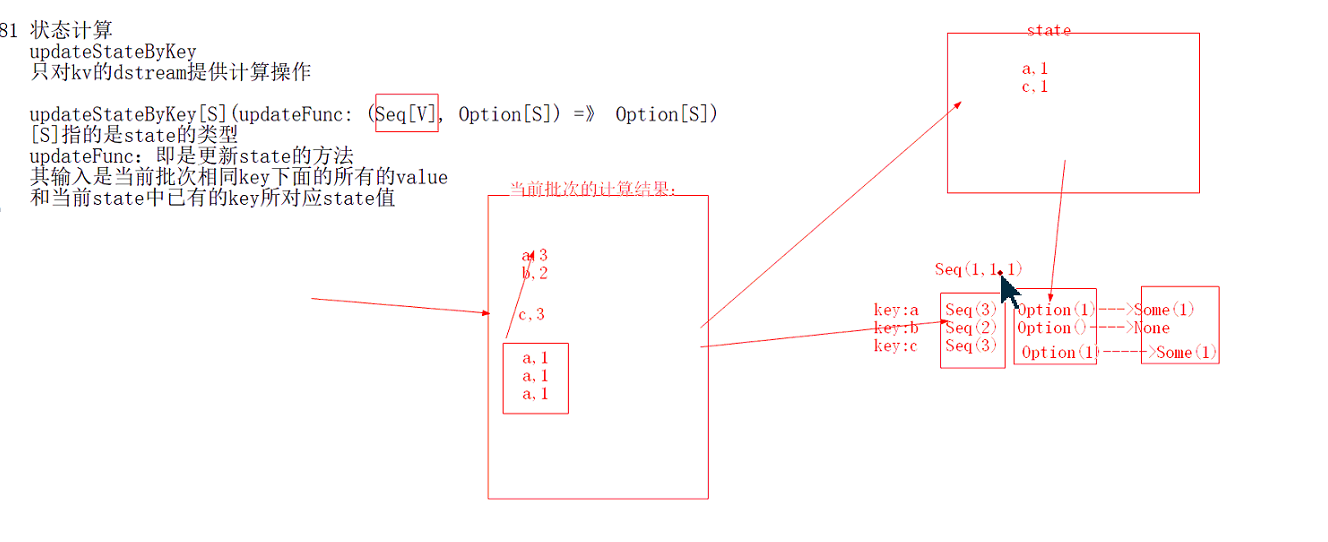


## 状态计算

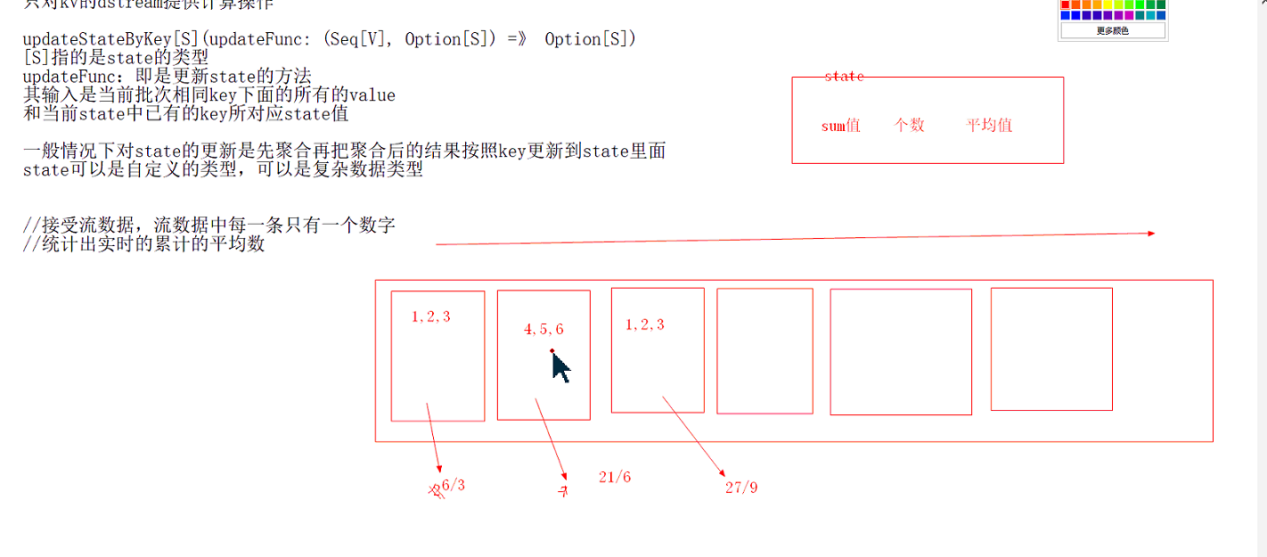
updateStateByKey

只对kv的dstream提供计算操作

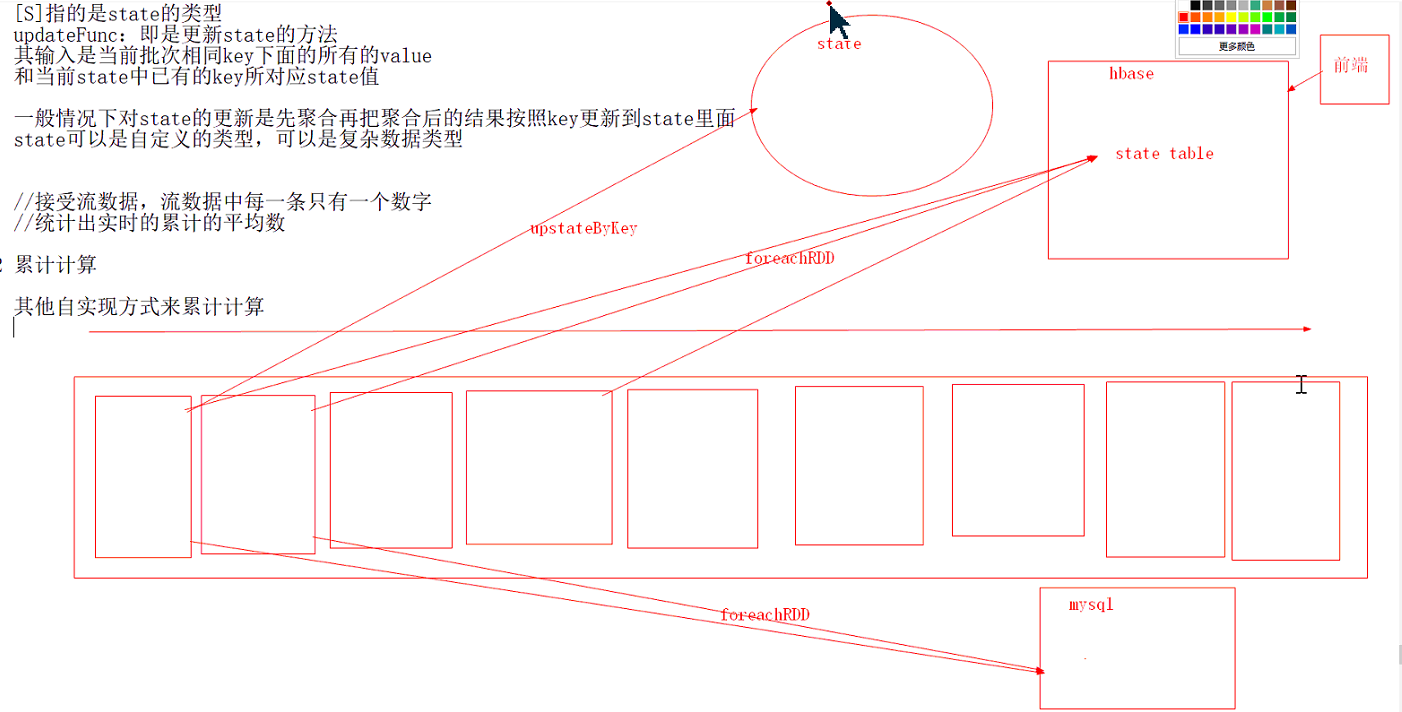




作业：



|  |
| --- |
| **package** top.xiesen.bd14.homework  **import** org.apache.hadoop.hbase.client.Put **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** top.xiesen.bd14.result.HbaseUtils  **object** StreamingAvg {   **val** *conf* = **new** SparkConf().setMaster(**"local[\*]"**).setAppName(**"StreamingAvg"**)  **val** *ssc* = **new** StreamingContext(*conf*, *Seconds*(3))  *ssc*.checkpoint(**"/temp/streamingavg"**)   **val** numAccumulator = *ssc*.sparkContext.longAccumulator(**"num"**)   *// 统计累计的平均值(接收的数字,总共多少个数字)* **def** savg() = {  **val** dstream = *ssc*.socketTextStream(**"master"**, 8899)  **val** result = dstream.map(x => {  numAccumulator.add(1)  (0, x.toInt)  }).reduceByKey(\_ + \_)   **val** state = result.updateStateByKey((nowstate: Seq[Int], num: Option[Int]) => {  num **match** {  **case** *Some*(value) => *Some*(value + nowstate.sum)  **case** None => *Some*(nowstate.sum)  }  })   *// 保存结果到hbase  // create 'streamingavg','i' 将总和作为columequality,将平均值作为value，* state.map(x => {  **val** table = HbaseUtils.*getTable*(**"streamingavg"**)  **val** put = **new** Put(**"rowkey"**.getBytes())  put.addColumn(**"i"**.getBytes(), **"avg"**.getBytes(), (x.\_2 \* 1.0 / numAccumulator.value).toString.getBytes())  put.addColumn(**"i"**.getBytes(), **"sum"**.getBytes(), x.\_2.toString.getBytes())  put.addColumn(**"i"**.getBytes(), **"num"**.getBytes(), numAccumulator.value.toString.getBytes())  table.put(put)  (x.\_2 \* 1.0 / numAccumulator.value, x.\_2)  }).print()  }   **def** main(args: Array[String]): Unit = {  *savg*()  *ssc*.start()  *ssc*.awaitTermination()  } } |



工具类

|  |
| --- |
| **package** top.xiesen.bd14.result  **import** org.apache.hadoop.hbase.{HBaseConfiguration, TableName} **import** org.apache.hadoop.hbase.client.{ConnectionFactory, Get} **import** org.apache.hadoop.hbase.util.Bytes  */\*\*  \* Hbase工具类  \*/* **object** HbaseUtils {   *// 创建连接* **val** *conf* = HBaseConfiguration.*create*()  **val** *connection* = ConnectionFactory.*createConnection*(*conf*)   *// 获取table对象用来对数据crud* **def** getTable(tableName: String) = {  *connection*.getTable(TableName.*valueOf*(tableName))  }   *// 获取对应表中的value值* **def** getValue(tableName: String, rowkey: String, family: String, qualifier: String) = {  **val** table = *getTable*(tableName)  **val** get = **new** Get(rowkey.getBytes())  get.addColumn(family.getBytes(), qualifier.getBytes())  **val** result = table.get(get)  Bytes.*toString*(result.value()  )  }  } |

累加结果保存到hbase中

|  |
| --- |
| **package** top.xiesen.bd14.homework  **import** org.apache.hadoop.hbase.client.Put **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** top.xiesen.bd14.result.HbaseUtils  **object** ReplaceState {   **val** *conf* = **new** SparkConf().setAppName(**"ReplaceState"**).setMaster(**"local[\*]"**)  **val** *ssc* = **new** StreamingContext(*conf*, *Seconds*(3))   **def** main(args: Array[String]): Unit = {  **val** dstream = *ssc*.socketTextStream(**"master"**, 9988)  **val** result = dstream.flatMap(\_.split(**"\\s"**))  .map((\_, 1))  .reduceByKey(\_ + \_) *// result.print()   // create 'state\_wc','i' rowkey = word; columequality =num ;value=wordnum* result.foreachRDD((rdd, time) => {  rdd.foreachPartition(wcs => {  **val** table = HbaseUtils.*getTable*(**"state\_wc"**)  **for** (wc <- wcs) {  **val** value = HbaseUtils.*getValue*(**"state\_wc"**, wc.\_1, **"i"**, **"num"**)  **val** put = **new** Put(wc.\_1.getBytes())  **if** (value == **null**) {  put.addColumn(**"i"**.getBytes(), **"num"**.getBytes(), wc.\_2.toString.getBytes())  } **else** {  put.addColumn(**"i"**.getBytes(), **"num"**.getBytes(), (wc.\_2 + value.toInt).toString.getBytes())  }  table.put(put)  }  })  })  *ssc*.start()  *ssc*.awaitTermination()  } } |